DOI:10.1145/2347736.2347759

A Fast Solver for a Class of Linear Systems

By Ioannis Koutis, Gary L. Miller, and Richard Peng

Abstract

The solution of linear systems is a problem of fundamental theoretical importance but also one with a myriad of applications in numerical mathematics, engineering, and science. Linear systems that are generated by real-world applications frequently fall into special classes. Recent research led to a fast algorithm for solving symmetric diagonally dominant (SDD) linear systems. We give an overview of this solver and survey the underlying notions and tools from algebra, probability, and graph algorithms. We also discuss some of the many and diverse applications of SDD solvers.

1. INTRODUCTION

One of the oldest and possibly most important computational problems is that of finding a solution to a system of linear equations. There is evidence that humans have been solving linear systems to facilitate economic activities since at least the first century AD. With the advent of physical sciences and engineering, linear systems have been, for centuries now, a central topic of applied mathematics. And over the last two decades, the digital revolution has expanded the use of linear system solvers to applications of surprising variety.

Many of these new applications typically model entities and their relationships as networks, also known as graphs, and use solvers to extract information from them. The resulting linear systems frequently obey simple constraints which classifies them as **symmetric diagonally dominant** (SDD).

An example of an area where such systems arise is in the analysis of social networks. Such networks can be represented as a set of links connecting people; an example is shown in Figure 1. A natural question to ask is how "close" are two persons in the network. Purely graph-based methods measure either the length of the shortest path or the maximum number of disjoint paths between the two nodes, but not both. To take both of these quantities into account we can view the network as an electric circuit with each connection corresponding to an electrical wire. Hooking a battery at the two nodes of interest and measuring the resistance of the entire network gives a quantity known as the **effective resistance**, which can be used as a "proximity" measure. Since the electrical network is not physically available, we cannot measure the effective resistance. We can, however, compute it by solving an SDD linear system.

The above example is only one of many instances of inference on a graphical model. Similar methods are applicable in a wide range of problems, such as measuring the importance of specific proteins in protein–protein interaction networks[14]; the link prediction problem in social networks[13]; or even problems where graphs arise less directly, such as segmenting the image shown in Figure 2.

More intricate uses of electrical networks have been discovered in the context of classical graph optimization problems, with the recent network flow algorithm by Christiano et al.[5] standing out as an algorithmic breakthrough. The algorithms reduce the problem to not just one network,

Figure 1. Representing a social network as a graph.

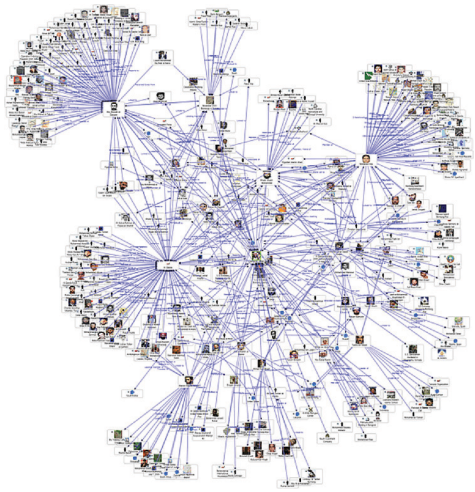
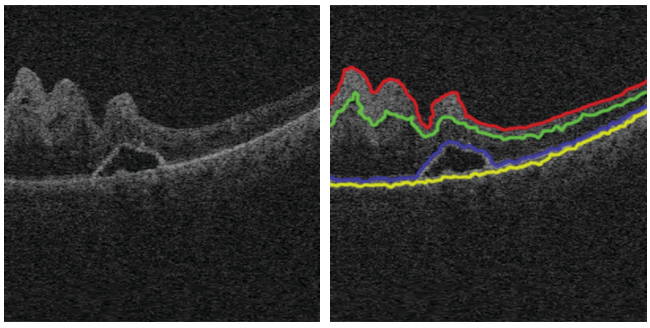


Figure 2. Segmentation of medical scans.[21]



This paper is based on two previous works: "Approaching Optimality for Solving SDD Linear Systems," which appeared in the *Proceedings of FOCS '10* and "A Nearly-*m* log *n* Time Solver for SDD Linear Systems," which appeared in the *Proceedings of FOCS '11*.

but to a sequence of networks via successive readjustment of edges. In these algorithms, some of the resulting systems are significantly harder than "typical" instances, capturing—in some sense—the hardness of the optimization problems themselves.

Current SDD solvers are empirically fast for some engineering applications, but they are not able to efficiently solve most cases of SDD linear systems. Besides these practical limitations, the fact that existing SDD solvers lack guarantees on arbitrary instances limits their implications to the theory of algorithms as well.

These factors underline the need for "mathematically certified" solvers that are **provably** fast for arbitrary SDD linear systems, independently of their origin, be it—for instance—social or protein networks. This paper describes our state of the art solver for SDD linear systems.

1.1. A glimpse at the solver

The class of SDD linear systems arises in particular in the study of electrical networks, which provide us with a concept crucial to understanding how our algorithm works: the effective resistance between two points in a network. In addition, the connection to networks enables adopting a second alternative view of our linear system, as a **weighted graph**. We give the details in Section 2.

We then move to the **algebraic component** of our solver. The approximate solution of linear systems via iterative methods is a topic not commonly encountered in computer science but thoroughly studied in the context of numerical linear algebra and scientific computation. Section 3 explains iterative methods via an analogy with the computation of the inverse of a real number in a calculator with a broken division key, where only addition and multiplication operations are available. This leads us to preconditioning, a term first used by Alan Turing. In the graph theoretic context, preconditioning provides a measure of **similarity** between graphs. This measure is used to formalize **design conditions** sufficient for the construction of a fast iterative method.

What distinguishes our solver from classical iterative methods is its **combinatorial component** and specifically the use of graph theoretic algorithms. It was understood before our work that the key to a fast solver is finding a subgraph (the preconditioner) which is similar to a given graph, but has substantially fewer edges.[20] Our contribution is a conceptually simple procedure for constructing good preconditioners, this is the topic of Section 4.

The base of our preconditioner is a spanning tree of the input graph, in other words a minimally connected subgraph. Our algorithm needs a special type of spanning tree called a low-stretch tree (LSST) which we describe in Section 4.1. This can be found using very sophisticated but fast algorithms.

To get a better preconditioner, we perform random sampling: each edge of the input graph is put into the preconditioner with a specified probability. It was known that the effective resistance between the two endpoints of each edge provides a good sampling probability for it.[18] Unfortunately the problem of computing the effective resistance seems to require solving an SDD linear system, which is the problem we are trying to solve in the first place.

Our main contributions are two ideas that allow us to circumvent this "chicken and egg" problem. The first idea is to use an upper estimate on the effective resistance for each edge. The second idea is to compute these estimates on a modified graph, in which the estimates are sufficiently good. The modification is in fact quite simple; we find an LSST of the graph and increase the weight all of its edges. To compute the upper estimate for the effective resistance of an edge in the modified graph we only use the edges of the LSST. A key side effect of this modification is that the number of non-tree edges in the preconditioner is much less than the number of edges in the original graph. In this way we meet the known design conditions and obtain the faster solver.

2. NETWORKS, SYSTEMS, SOLVERS

Let us consider the problem of finding a voltage setting given the desired net current flow at each of the vertices. A simple three-node example of an electric network is depicted in Figure 3. The inverse of the resistance of wire, also known as **conductance**, is a direct analogue to the edge weight in a graph; because of that we choose to label each wire by its conductance rather than its resistance. Setting the **voltages** of the vertices to some values leads to an **electrical flow** through the edges. There are two fundamental principles governing this voltage setting. (a) Kirchhoff's law, which states that with the exception of the vertices where current is injected/extracted, the net flow at each vertex is zero. (b) Ohm's law, which states that the current on an edge equals the voltage difference between its endpoints times the conductance of the wire.

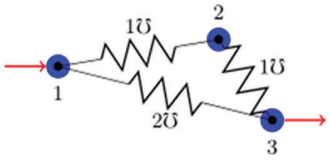
As an example consider the network given in Figure 3 where we set the voltages at the three vertices to be x_1, x_2, and x_3 respectively. By Ohm's law we get that the current flows along edges $1 \to 2$ and $1 \to 3$ are $1 \cdot (x_1 - x_2)$ and $2 \cdot (x_1 - x_3)$, respectively. Therefore the amount of current we will need to inject into vertex 1 to maintain these voltages is:

$$1 \cdot (x_1 - x_2) + 2 \cdot (x_1 - x_3) = 3x_1 - x_2 - 2x_3$$

Identities for the required current entering/leaving vertices 2 and 3 can also be derived similarly. Therefore, if we want one unit of current to enter at vertex 1 and leave at vertex 3, the voltages will need to satisfy the following system of linear equations:

Using more compact notation, linear systems assume the form $Ax = b$ where x is a $n \times 1$ column vector of unknowns,

Figure 3. A simple network and linear system.



$$\begin{aligned} 3x_1 - 1x_2 - 2x_3 &= 1 \\ -1x_1 + 2x_2 - 1x_3 &= 0 \\ -2x_1 - 1x_2 + 3x_3 &= -1 \end{aligned}$$

also called variables, b is a $n \times 1$ column vector of real numbers, and A is an $n \times n$ matrix containing the coefficients of the variables. For example, the above linear system can be expressed in matrix form as:

$$\begin{bmatrix} 3 & -1 & -2 \\ -1 & 2 & -1 \\ -2 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}. \quad (2.1)$$

Note that each off-diagonal entry is the negation of the conductance of the resistor connecting its two vertices, and each diagonal entry is the sum of the conductances of all resistors incident to the corresponding vertex. Since resistive networks can also be viewed as undirected graphs, this type of matrix is known as a **Graph Laplacian** and we will rely on this connection extensively in our algorithm. SDD matrices are a further generalization of graph Laplacians. However, an SDD system can be easily transformed into a Laplacian system (e.g. see Koutis et al.[9]) and so we will restrict our attention entirely to graph Laplacians.

Once we're able to obtain the voltages at each vertex, we can also compute the **effective resistance** between two vertices. Intuitively, this notion can be viewed as thinking of the entire network as a single electrical component. Then by Ohm's law the voltage drop required to send 1 unit of current corresponds to the resistance of the component. In our example, the effective resistance between vertex 1 and 2 is $x_1 - x_3 = 2/5$. Formally, this value equals $v_s - v_t$ from the solution of the linear system $Lv = \varphi$, where φ is zero everywhere except in the two entries corresponding to the nodes s and t, for which we set $\varphi_s = 1$ and $\varphi_t = -1$. As we will see later, this metric is not only used for network analytics, but also plays a crucial role in our solver itself.

2.1. Solvers and their speed

Despite its long history, the problem of constructing good solvers is considered far from being solved, especially in terms of speed. The speed of algorithms is commonly measured in terms of the input size. In the case of general linear systems on n variables, the matrix has size n^2. However, matrices are often **sparse**, that is, most of their entries are equal to zero. Because of this we can easily "compress" them to size proportional to the **number of non-zeros**, denoted by m. The best case scenario, which remains entirely consistent with our current understanding, is that linear systems can be solved with $O(m)$[a] operations.

It's fair to say that Gaussian elimination is the most well-known method for solving linear systems. It runs in $O(n^3)$ time and it is known as a direct method in that, if the arithmetic operations are performed exactly then one gets the exact solution to the system. Although this exponent of 3 has been decreased to as low as 2.37,[24] direct methods in general require storing n^2 entries, creating a natural bottleneck that limits us to systems with a few thousand variables.

One possible remedy to the space and time limitations of direct methods are **iterative methods**. These compute progressively better approximate solutions by only performing **matrix–vector products** and other simpler vector operations.

One of the most important iterative methods is Conjugate Gradient, discovered by Lanczos, Hestenes, and Stiefel in the early 1950s. This method works for arbitrary symmetric positive definite systems, a class that includes SDD systems. While it requires only $O(m)$ space, it is understood that its running time—in its original form—can be large.

Strong evidence that iterative methods can combine low space requirements and very fast running time was provided by a family of iterative methods known as multigrid.[22] Multigrid solvers have an $O(m)$ running time guarantee albeit for restricted and well-structured systems that arise in scientific computing.

The solver we will review in this paper is also an iterative method. It is the culmination of a line of work initiated by Vaidya,[23] which was brought to near-completion with the breakthrough achievement of Spielman and Teng[19]: the first solver that runs in time $O(m \log^c n)$ for any graph Laplacian, where c is a large constant. The work discussed here, summarized in the following claim from Koutis et al.,[10] provides a conceptually simpler, faster and more practical algorithm.

THEOREM. SDD systems can be solved in $\tilde{O}(m \log n \log(1/\epsilon))$ time,[b] where ϵ is a standard measure of the approximation error.

3. THE ALGEBRAIC COMPONENT

3.1. Iterative methods: Division-free inversion

Our way towards the faster solver starts with a basic and perhaps seemingly unrelated question: is it possible to compute the inverse $1/\alpha$ of a number α using a calculator with a broken division key?

To answer the question we can invoke a basic identity that tells us that when $0 < \alpha < 2$, $1/\alpha$ equals the following infinite sum:

$$\begin{aligned} 1/\alpha &= 1/(1-(1-\alpha)) \\ &= 1 + (1-\alpha) + (1-\alpha)^2 + (1-\alpha)^3 + \ldots \end{aligned} \quad (3.2)$$

Of course, computing an infinite sum is not possible. But keeping a number of terms will give us an **approximation** of $1/\alpha$; the more terms we keep the better the approximation.

But how is this related to the problem of solving linear systems? Matrices borrow several of the usual properties of scalar numbers. When A is symmetric, its inverse A^{-1} also satisfies the identity in 3.2, substituting A for α, A^{-1} for $1/\alpha$ and the identity matrix I for the number 1. Furthermore, if we want an approximation to $x = A^{-1}b$[c] we can actually avoid entirely taking powers of the matrix; the ith approximate vector

$$x^{(i)} = (I + (I - A) + \ldots (I - A)^i)b$$

[a] We use $f(n) = O(g(n))$ to denote $f(n) \le c \cdot g(n)$ when $n \ge n_0$ for some constants c and n_0.

[b] The $\tilde{O}()$ notation hides a log log n factor.

[c] If A^{-1} does not exist, as in the case of Laplacians, we use A^{-1} to denote the pseudoinverse as well.

can be produced with i applications of the following simple recurrence:

$$x^{(0)} = 0$$
$$x^{(i+1)} = b + (I - A)x^{(i)} \text{ for } i > 0.$$

It can be seen that each step involves a matrix–vector multiplication by A. This is the simplest among **iterative** methods that in general attempt to approximate the solution of a linear system using only a sum of results from a series of matrix–vector multiplications.

3.2. Preconditioning

So far, our replacement for the division button is of rather restricted value, since it only works when $0 < \alpha < 2$, and can converge very slowly when α is close to 0 or 2. One way to extend our method and to speed up its convergence is to add a "restricted division" key to our calculator. This key allows us to "divide" by a fixed scalar β of our choice, which in the matrix setting corresponds to a matrix–vector product involving the inverse, B^{-1} of a matrix B. We can speed up our algorithm by pressing the "restricted division" button after each matrix–vector multiplication by A, giving the following modified recurrence known as **preconditioned Richardson iteration**:

$$x^{(0)} = 0$$
$$x^{(i+1)} = B^{-1}b + (I - B^{-1}A)x^{(i)} \text{ for } i > 0.$$

The matrix B is known as the **preconditioner** and instead of solving the system $Ax = b$, we are essentially solving the **preconditioned** system: $B^{-1}Ax = B^{-1}b$. It is worth emphasizing that each step of this method involves a matrix–vector multiplication by A followed by a "division" by the matrix B.

3.3. Measuring similarity between matrices

Looking back at the single variable recurrence, the critical condition for its convergence is $0 < \alpha < 2$. An extension of it is needed in order to analyze preconditioned iterative methods involving matrices A and B. For matrices A and B, we say $A \preceq B$ when for all vectors x we have

$$x^T Bx \leq x^T Ax.$$

Unlike the situation with scalars, this ordering is only "partial". Even for size 2 diagonal matrices, it is possible that neither $B \preceq A$ nor $A \preceq B$ holds. But when A and B are symmetric, there will be numbers κ_{max} and κ_{min} such that:

$$\kappa_{min} A \preceq B \preceq \kappa_{max} A.$$

We will say that B is a κ-approximation of A, where $\kappa = \kappa_{max}/\kappa_{min}$. In this case, after introducing additional scaling factors, it can be shown that the preconditioned Richardson's iteration gives a good approximation in $O(\kappa)$ iterations. There are iterative methods with faster convergence rates, and—as we will see—our solver relies on one of them, known as Chebyshev iteration.

3.4. Interpreting similarity

It is interesting to see what this measure of similarity means in the context of electrical networks, that is when both A and B are Laplacians. The quadratic form

$$x^T Ax$$

is equal to the **energy dissipation** of the network A, when the voltages at vertices are set to the values in the vector x. Then, the network B is a κ-approximation of the network A whenever for all voltage settings, B dissipates energy which is within a κ factor of that dissipated by A.

So, roughly speaking, **two networks are similar when their "energy profiles" are similar**. This definition does not necessarily correspond to intuitive notions of similarity; two networks may appear to be very different but still be similar. An example is shown in Figure 4.

3.5. What is a good preconditioner?

Armed with the measure of similarity, we are now ready to face the central problem in solver design: how do we compute a good preconditioner?

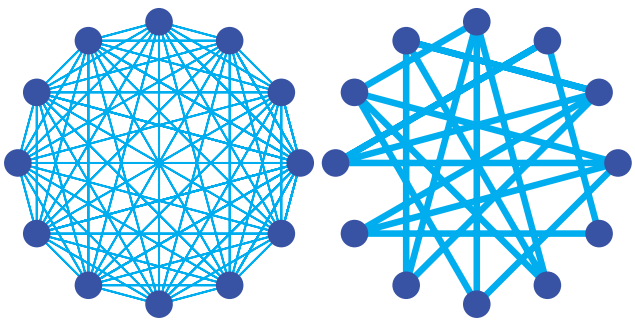
To deal with the question we must first understand what properties are desirable in a preconditioner. A big unknown in the total running time is the cost incurred by the limited division button that evaluates $B^{-1}y$.

To evaluate $B^{-1}y$ we do not need to compute B^{-1}. We can instead solve the system $Bz = y$; the solution z will be equal to $B^{-1}y$. Clearly, we would like to solve systems involving B as quickly as possible. At the same time we would like the number of iterations to be as small as possible, since each of them requires at least m operations. Furthermore, a slow algorithm for computing the preconditioner B would defeat the purpose of a fast solver. So, we should also be able to find B quickly. Balancing these three opposing goals makes the problem quite challenging.

3.6. Recursion and design conditions

In order to solve the linear system fast, we will need a preconditioner B which is an **extremely good** approximation of A **and** can be solved in linear time. Satisfying both these requirements is too much to hope for. In practice, any good graph preconditioner B won't be significantly easier to solve

Figure 4. Two similar graphs: A complete graph and a random small subset of its edges, made heavier.



comparing to A. As a result, there is no hope that preconditioned Richardson's iteration or any other preconditioned method can lead to fast solvers.

The remedy to the problem is **recursion**. In a recursive preconditioned method, the system in the preconditioner B is not solved exactly but approximately, via a recursive invocation of the same iterative method. We now have to find a preconditioner for B, and furthermore a preconditioner for it and so on. This produces a **multilevel hierarchy** of progressively smaller graphs.

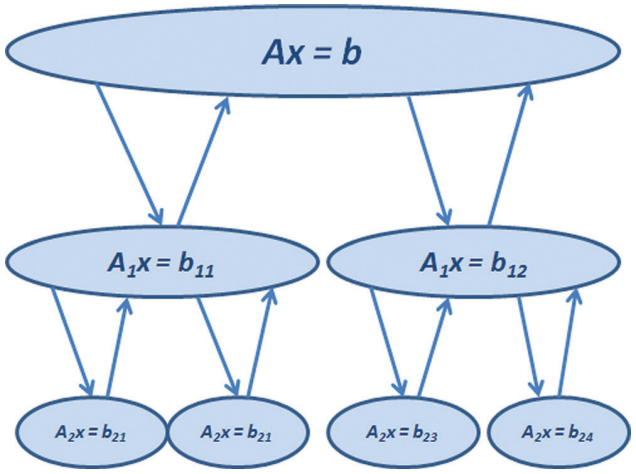
Rohklin, Spielman, and Teng[19] analyzed a recursive iterative method which moves between levels of the hierarchy as shown in Figure 5; for each visit at level i, the algorithm makes k visits to level $i+1$. Every time the algorithm returns to the ith level it performs **matrix–vector multiplications** with the graph A_i, and other simpler operations; so the work is proportional to the number of edges of A_i. To keep the **total work** as small as possible, that is $O(km)$, the graphs in the hierarchy must get smaller sufficiently fast. In particular, it is sufficient that the graph on level $i+1$ is smaller than the graph on level i by a factor of $1/(2k)$.

However, the algorithm must converge within the $O(km)$ time bound. To achieve this, the iterative method analyzed within this recursive framework is a method known as Chebyshev iteration. It requires only $O(k)$ iterations, when B is a k^2-approximation of A, as compared to the $O(k^2)$ iterations required by Richardson's iteration. Using this fact Spielman and Teng arrived at **design conditions** that are sufficient for a fast solver.[20] It was actually shown that a good algorithm for preconditioning **extends** to a good solver. More specifically, assume that for some fixed value C and any value of k, we have a fast algorithm that given A, produces a k^2-approximation with $n + C \cdot m/k$ edges. Then we automatically get a solver that runs in time $O(k \cdot m)$.

Carefully checking the above statement, we realize that there is a slight discrepancy. If m is close to n and k is large, then $n + C \cdot m/k$ will be bigger than m, which contradicts our promise for a multilevel hierarchy of progressively smaller graphs. However, as observed by Vaidya,[23] when m is almost the same n, the graph has several "tree-like" parts, and these can be reduced via a "partial" Gaussian elimination that runs in $O(m)$ time. So whenever this case appears, it makes sense to first run partial elimination. This will decrease the vertex count n, leading to a much smaller instance on which recursion is applicable.

The multilevel analysis of Spielman and Teng is significant not only for its actual algorithmic value but also the conceptual **reduction** of the multi-level solver design problem to a well-defined two-level preconditioning problem, allowing us now to focus on the combinatorial component of the solver.

Figure 5. The sequence of calls of a recursive iterative method. The matrix is fixed at each level.



4. THE COMBINATORIAL COMPONENT

Although graph theory has been used to speed up direct methods, it took a paradigm-shifting idea of Pravin Vaidya to enter a **systematic study** of using graph theory for iterative methods. In particular, Vaidya suggested the use of a **spanning tree** of the graph A as a **building base** for the preconditioner B. A spanning tree of a graph is a connected subgraph without loops. The choice of a tree stems from the observation that linear systems whose matrix is the Laplacian of a tree can be solved in $O(n)$ time via Gaussian elimination. Adding a few edges of A back onto the tree returns a preconditioner B which can only be better than the tree, while still being relatively easy to solve. Vaidya's idea set forth two questions: (i) What is an appropriate base tree? (ii) Which off-tree edges should be added into the preconditioner?

While these questions seem to be interrelated, we can actually address them separately.

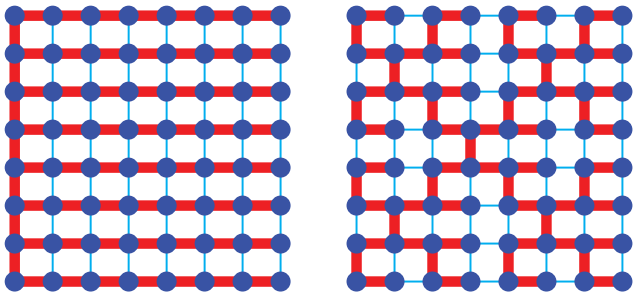
4.1. Low-stretch: The base spanning tree

The goal of finding a preconditioning tree B which is as similar as possible to the graph A led Vaidya to a natural idea: use a tree which concentrates the maximum possible weight from the total weight of the edges in A.

The maximum-weight spanning tree idea led to the first non-trivial results, but does not suffice for our algorithm. In fact, the weight measure does not distinguish trees in unweighted graphs, where all trees have equal weight.

The key to finding a good tree to use as a building base is the notion of **stretch**: For every edge (u, v) of the graph, there is a unique "detour" path between u and v in a tree T. The stretch of the edge with respect to T is equal to the distortion caused by this detour, and in the unweighted case, it is simply the length of the tree path. This notion generalizes naturally to the weighted case, which we will formalize in Section 4.3. The **total stretch** of a graph A with respect to a tree T is the sum of the stretches of all the off-tree edges. A **low-stretch tree** (LSST) is one for which we have a good upper bound on the total stretch. So, at a high level, an LSST has the property that it provides good (on average) "detours" for edges of the graph. A concrete example on a larger unweighted graph is given in Figure 6, where the tree on the right has lower total stretch, and as it turns out is a better base tree to add edges to.

Figure 6. Two possible spanning trees of the unweighted square grid, shown with red edges.



Trees for the grid: a quantitative example

The two spanning trees of the 8×8 grid shown in Figure 6 can guide our understanding of the general $\sqrt{n} \times \sqrt{n}$ grid.

In the tree on the left, for each vertical edge beyond column $\sqrt{n}/2$, at least $\sqrt{n}$ horizontal edges are needed to travel between its endpoints; that means that its stretch is at least $\sqrt{n}$. So the $n/2$ edges in the right half of the square grid contribute a total stretch of $n^{1.5}$.

In the tree on the right, all edges along the middle row and column still have stretch $O(\sqrt{n})$. However, the middle row and column only have $O(\sqrt{n})$ edges and so they contribute only $O(n)$ to the total stretch. Recall that all we need is a low total stretch, so a small number of high-stretch edges is permitted. Having accounted for the edges in the middle row and column, the argument can then be repeated on the four smaller subgraphs of size $n/4$ formed by removing the middle row and column. These pieces have trees that are constructed similarly, leading to the recurrence

$$\text{TotalStretch}(n) = 4 \times \text{TotalStretch}(n/4) + O(n).$$

Its solution is $\text{TotalStretch}(n) = O(n \log n)$.

A generalization of this type of "accounting", that keeps the number of high stretch edges small, forms the basis of the current state-of-the-art algorithms.[1]

Algorithms for the computation of LSSTs were first studied in an unrelated context,[2] where it was shown that any graph contains a spanning tree with total stretch $O(m^{1+\epsilon})$; the tree can be found in $O(m \log n)$ time. The total stretch was lowered to $O(m \log^2 n)$ in Elkin et al.,[6] and further to $\tilde{O}(m \log n)$ in Abraham and Neiman,[1] giving the following theorem.

THEOREM. Every graph has a spanning tree of total stretch $\tilde{O}(m \log n)$. The tree can be found in $\tilde{O}(m \log n)$ time.

Boman and Hendrickson first introduced LSSTs as stand alone preconditioners in 2001. This was a catalyst to subsequent progress, which used LSSTs as a base for building even more intricate preconditioners. In fact, LSSTs are indispensable components of all nearly-linear time SDD system solvers. It is worth pointing out that while LSSTs were originally conceived as potentially good two-level preconditioners, their full power in the context of multilevel solvers was not realized until our work, which we describe in Section 4.3.

4.2. Sparsification

Spielman and Teng's[19] main contribution was a "tour de force" algorithm for finding a preconditioner that's the LSST plus a small number of edges. It took many by surprise as it yielded the first nearly-linear time SDD solver.

Describing the algorithm is out of the scope of this article, but it is worth noting its two enhancements over previous approaches. First, instead of just adding off-tree edges from A back onto the tree, the algorithm **re-weights** them. The tree edges may themselves be re-weighted in the final preconditioner B. Second, the procedure for adding edges in B is not deterministic but **randomized**, as it contains a process for sampling edges from A.

However the major conceptual and technical contribution of Spielman and Teng that formed the cornerstone of their solver was a **sparsification** algorithm. They showed that every graph A has a 2-approximation B which has $O(n \log^c n)$ edges for some large constant c. The graph B is called the sparsifier and, of course, it can be used as a preconditioner when A is dense.

After the first sparsification result, progress towards faster SDD solvers took a detour through the study of spectral sparsification as a stand-alone problem. Works by Batson, Kolla, Makarychev, Saberi, Spielman, Srivastava, and Teng led to nearly-optimal spectral sparsifiers, albeit at the cost of much higher running time. These results were motivated by the work of Spielman and Srivastava,[18] who gave an extremely simple algorithm for finding spectral sparsifiers with only $O(n \log n)$ edges. Their algorithm, as well as the Spielman–Teng spectral sparsification algorithm builds upon a framework established by Benczur and Karger for sampling and re-weighting a graph.

The framework requires positive numbers t_e assigned to each edge, corresponding to the relative probabilities of sampling them. It calculates the sum of these numbers, $\sum_e t_e$ and proceeds for $O(t \log n)$ rounds. In each round one new edge is added to the sparsifier B. The edge is picked randomly with replacement among the m edges of A, but not in a "fair" way. An edge e is picked with relative probability t_e, which equates to a probability of $p_e = t_e/t$. Once an edge is picked, it is added to B with weight scaled down by a factor of $O(t_e \log n)$. Furthermore, if an edge is picked twice or more during this process, each new copy is added as a parallel edge, making B potentially a multi-graph.

Understanding re-weighting

While it may appear complicated, the re-weighting choice is quite natural. The reasoning is that the "expected value" of B should be A itself on an edge-to-edge basis. In other words, the average of many B's output by the algorithm should be A itself.

Spielman and Srivastava found the probabilities that give sparsifiers with the fewest number of edges with the help of some experimentation. Amazingly, the answer turned out to be related to the effective resistance of the edge, specifically $t_e = w_e R_e$. With hindsight, it is interesting to reflect about the natural meaning of effective resistance. If there is a wire of resistance $r_e = 1/w_e$, between i and j, the effective resistance R_e will in general be smaller than r_e because most probably there will be other network connections to accommodate the flow; this is known as Rayleigh's monotonicity theorem. The extreme case $w_e R_e = 1$ occurs only when there is no other route between i and j except the wire joining them. In this situation, the edge (i, j) is crucial for the network. On the other hand if $w_e R_e$ is very small, there must be significant alternative network connections between (i, j). Therefore, the product $w_e R_e$ as a measure of the importance of a wire. Using tools from modern matrix theory,[15] Spielman and Srivastava proved that this algorithm does return a good spectral sparsifier with high probability. Combining with the fact that $t = \sum_e w_e R_e = n - 1$ yields the overall number of edges: $O(n \log n)$.

Despite being a major improvement in the theory of graph sparsification, the algorithm of Spielman and Srivastava did not accelerate the SDD solver as current methods for quickly computing effective resistances require the solution of linear systems. The guarantee of $O(n \log n)$ edges is also hard to connect with the $n + C \cdot m/k$ edges needed by the design condition. However, it is fair to say that their result cleared the way to our contribution to the problem.

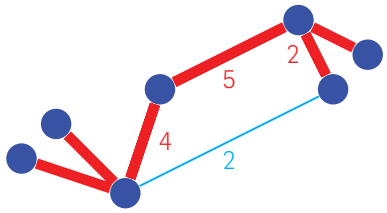
4.3. Which off-tree edges?

If we cannot effectively compute effective resistances, can we at least approximate them quickly, even poorly? A closer look at the matrix concentration bounds allows us to relax this goal a bit further: the sampling algorithm described in Section 4.2 can be shown to work with any choice of t_e, as long as $t_e \geq w_e R_e$. The observant reader may notice that the expected number of times e is picked is $O(t_e \log n)$, so increasing t_e only results in more copies of e being picked without affecting the expectations of all other edges.

The intuition that the low-stretch spanning tree must be part of the preconditioner leads us to taking tree-based estimates R'_e for the effective resistances R_e. In particular, for an off-tree edge e we let R'_e be the sum of the resistances along the unique path between the endpoints of e in the tree, as shown in Figure 7. By Rayleigh's monotonicity theorem, we know that this estimate will be higher than the actual R_e. This leads to the tree-based sampling probability for an off-tree edge e being proportional to $t_e = w_e R'_e$. Furthermore, if we keep the entire tree in B, we can modify the sampling algorithm presented in Section 4.2 to only sample off tree edges. Then the total number of off-tree (multi) edges sampled in B is $O(t \log n)$ where t is the sum of all t_es, which in turn. This brings us to the question: how big is t?

This question leads us back to the discussion of low-stretch spanning tree and the definition of **stretch** for the general weighted case: if we view the length of an edge e as the inverse of its weight, then its stretch equals

Figure 7. The effective resistance R'_e of the blue off-tree edge in the red tree is 1/4 + 1/5 + 1/2 = 0.95. Its stretch $w_e R'_e$ is (1/4 + 1/5 + 1/2)/(1/2) = 1.9.



to $w_e R'_e$.[d] Therefore, t is the **total stretch** of the off-tree edges with respect to the tree. Then, using the low-stretch spanning tree of the Theorem in Section 4.1, we can upper bound t by $O(m \log n)$. Recall that the number of samples will be $t \log n$ and so it appears that we do not gain much from the sampling process unless the graph A has a very special tree.

Our key idea is to make a special graph $\tilde{A}$ out of A. We do so by **scaling up**, by a factor of κ, the weights of edges of a low-stretch spanning tree in A. For an edge that's not part of the tree, its weight does not change, while the tree path connecting its endpoints is now heavier by a factor of κ. So the stretch decreases by a factor of κ and the total stretch of these edges becomes $t = O((m \log n)/\kappa)$. Now, consider what happens if we sample the off-tree edges in $\tilde{A}$. The output B will be a 2-approximation of $\tilde{A}$. On the other hand, the graph $\tilde{A}$ is a κ-approximation to A, and by transitivity B is 2κ-approximation to A. Also, the number of non-tree edges sampled will be $O(t \log n) = O((m \log^2 n)/\kappa)$. Adding in the $n - 1$ tree edges gives a total of $n + O((m \log^2 n)/\kappa)$ edges in B. Recall that the two-level **design conditions** stated in Section 3.6 require a k^2-approximation with $n + C \cdot m/k$ edges in order to obtain a running time of $O(k \cdot m)$. So by setting κ to $O(\log^4 n)$, we meet the conditions with $k = O(\log^2 n)$ and arrive at our first result:

THEOREM[9]. SDD systems can be solved in $\tilde{O}(m \log^2 n \log(1/\epsilon))$ time, where ϵ is a standard measure of the approximation error.

As it turned out, the low-stretch spanning tree is not only a good base tree, but also tells us which off-tree edges should go to the preconditioner. Our faster, $O(m \log n)$ time algorithm will come via an even better understanding of the properties of the tree.

4.4. The final push: Low-stretch spine

Assume that we are given a graph A, found its LSST T_A, and based on it, computed the preconditioner B. Then the $O(m \log^2 n)$ time solver algorithm dictates that we recursively do the same with B. But do we really have to scrap T_A and find another LSST T_B? After all, it may be the case that T_A is a LSST of B, or close to being one.

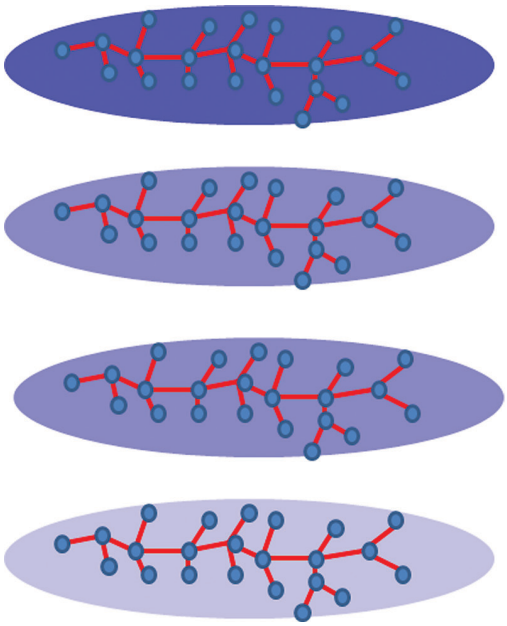
[d] An alternate view is that the stretch of e is the weighted length of the tree path between e's end points divided by e's own length.

What the $O(m \log^2 n)$ algorithm[9] missed is the observation that we can **keep sampling based on the same tree**, gradually generating **all** levels of the multilevel hierarchy, until what is left is the tree itself. This justifies thinking of a low-stretch spanning tree as a graph **spine**, and is depicted in Figure 8.

When the sparsifier B is viewed as a graph, it is possible for some of its edges to have high stretch. However, a more careful reexamination of the sampling algorithm shows that these edges are the result of an edge being sampled many times. From this perspective, these heavy edges are in fact many multi-edges, each with low stretch. Therefore, if we process these multi-edges separately, the tree T_A will be a low-stretch spanning tree in B, and the higher edge count is still bounded by the number of rounds made by the sampling algorithm. This allows us to use T_A as a low-stretch spanning tree and sample the off-tree edges in B according to it. Note that with this modification, it's possible for us to observe a temporary "slow down" in the reduction of the overall edge count; the preconditioner of B may have the same number of off-tree edges as B itself. However the total number of multi-edges will decrease at a rate that meets the design conditions. This reuse of the tree for generating sparsifiers is a crucial deviation from prior works.

But this doesn't fully explain the faster solver algorithm. To achieve it we need an extra trick. Assume for a moment that our graph A is what we call **spine-heavy**; that is, it has a tree of total stretch equal to $O(m/\log n)$. Then by an argument analogous to the one using a standard low stretch spanning tree, we can show that B actually satisfies the two-level preconditioning requirement for an even lower value of κ, namely a fixed constant. This, in combination with spine-based sampling allows us to solve **spine-heavy** graphs in **linear time**. A more global view of this algorithm, as shown in Figure 8 is that it progressively makes the tree heavier, while removing off-tree ones.

Figure 8. Low-stretch spanning tree as a spine. The "cloud" of off-tree edges becomes progressively sparser.



Since our initial graph A cannot be expected to be spine-heavy, we make a spine-heavy graph $\tilde{A}$ out of A, by scaling-up its LSST by an $O(\log^2 n)$ factor. Now $\tilde{A}$ is an $O(\log^2 n)$-approximation to A and we can solve it in $O(m)$ time. Using it as preconditioner for A completes the $O(m \log n)$ time solver. So, we have arrived at our destination.

THEOREM[10]. SDD systems can be solved in $\tilde{O}(m \log n \log(1/\epsilon))$ time, where ϵ is a standard measure of the approximation error.

5. EXTENSIONS

5.1. Parallelization

Several algorithms in numerical linear algebra have parallel versions that are work-efficient. A parallel algorithm is called **work-efficient** if it performs roughly the same work as its best sequential algorithm for the same problem, while allowing the use of parallel processing.

The first steps towards studying the parallelism potential of SDD solvers were taken in Blelloch et al.,[3] which presented a nearly (up to log factors) work-efficient algorithm, running in $O(m^{1/3})$ parallel time. Informally, this means that up to $m^{2/3}$ parallel processes can be used to accelerate the algorithm, a non-trivial potential for parallelism.

5.2. Implementation

The most complicated component of our solver is the algorithm for computing a LSST. It is however expected that a conceptually simpler algorithm for this problem is to be discovered, leading to a fast and "clean" implementation, and quite likely the removal of the log log n factors from the running time.

In a practical implementation, it would be a good idea to substitute the recursive preconditioned Chebyshev iteration by a recursive preconditioned Conjugate Gradient (PCG) iteration. It is known that, in two-level methods, PCG is essentially able to automatically optimize the performance of the preconditioner. It is expected that the same should be true for some multilevel variant of PCG, but this is yet to be proven.

We expect that, eventually, the best implementations of SDD solvers will combine ideas from this work and other existing graph-based techniques,[8] or entirely new ideas. Such ideas will certainly be needed to achieve—if possible—a "fully parallel", $O(\log n)$ time, work-efficient SDD solver.

6. THE LAPLACIAN PARADIGM

Solvers for SDD systems are increasingly viewed as an algorithmic **primitive;** a fundamental subroutine that can be used to design many other efficient algorithms. Indeed, since the Spielman–Teng breakthrough, the availability of fast SDD solvers has sparked what has been dubbed **the Laplacian paradigm**: an entire class of new algorithms spanning various areas. Because it is impossible to do justice to each one of these topics, we will present some unifying themes and only point to some representative examples of applications.

Perhaps the most direct example of using the solver as a primitive is the computation of eigenvectors. It was shown

in Spielman and Teng[20] that $O(\log n)$ iterations of solves produce a good approximation to a basic eigenvector of a graph. More closely related to preconditioned iterative methods is a solver for elliptic finite element linear systems.[4] This work showed that such systems can be preconditioned with graph Laplacians and so they can be solved in nearly linear time.

A more general framework stems from one of the most powerful discoveries in combinatorial optimization: interior point algorithms. It was shown by Daitch and Spielman[17] that interior point algorithms allow us to reduce a broad class of graph problems to solving a small number of SDD linear systems. This led to the best known running times for problems such as minimum cost flow and loss generalized flow. These problems are extensions of the maximum flow problem, which in its simplest version asks for the maximum number of edge disjoint routes (or "flow") between two nodes s and t. Further work in this direction led to the first improvement in 20 years on the approximate maximum flow problem.[5] The max-flow result is in turn directly applicable to graph partitioning, that is the separation of a graph to two well connected pieces; the fastest known algorithm for this problem repeatedly applies the fast max-flow algorithm.[16]

It is also worth noting that the solver presented in Blelloch et al.[3] readily gives—for all the above problems—parallel algorithms that are essentially able to split evenly the computational work and yield speedups even when only a small number of processors is available. This is a rare feature among previous algorithms for these problems.

Solver-based algorithms have already entered practice, particularly in the area of computer vision, where graphs are used to encode the neighboring relation between pixels. Several tasks in image processing, such as image denoising, gradient inpainting, or colorization of grayscale images, are posed as optimization problems for which the best known algorithms solve SDD systems.[11, 12] Linear systems in vision are often "planar", a class of SDD systems for which an $O(m)$ time algorithm is known.[7]

Given the prevalence of massive graphs in modern problems, it is expected that the list of applications, both theoretical and practical, will continue expanding in the future. We believe that our solver will accelerate research in this area and will move many of these algorithms into the practical realm.

Acknowledgments

This work is partially supported by NSF grant number CCF-1018463. I. Koutis is supported by NSF CAREER award CCF-1149048. Part of this work was done while I. Koutis was at CMU. R. Peng is supported by a Microsoft Research Fellowship. C

References

1. Abraham, I., Neiman, O. Using petal decompositions to build a low stretch spanning tree. In *Proceedings of the 44th Symposium on Theory of Computing* (STOC '12, 2012), ACM, New York, NY, 395–406.
2. Alon, N., Karp, R., Peleg, D., West, D. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput. 24*(1) (1995), 78–100.
3. Blelloch, G.E., Gupta, A., Koutis, I., Miller, G.L., Peng, R., Tangwongsan, K. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA '11, 2011), ACM, New York, NY, 13–22.
4. Boman, E.G., Hendrickson, B., Vavasis, S.A. Solving elliptic finite element systems in near-linear time with support preconditioners. *SIAM J. Numer. Anal. 46*(6) (2008), 3264–3284.
5. Christiano, P., Kelner, J.A., Mądry, A., Spielman, D., Teng, S.-H. Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*, 2011.
6. Elkin, M., Emek, Y., Spielman, D.A., Teng, D.A. Lower-stretch spanning trees. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, 494–503, 2005.
7. Koutis, I., Miller, G.L. A linear work, $O(n^{1/6})$ time, parallel algorithm for solving planar Laplacians. In *Proc. 18th ACM–SIAM Symposium on Discrete Algorithms (SODA)*, 2007.
8. Koutis, I., Miller, G.L. Graph partitioning into isolated, high conductance clusters: Theory, computation and applications to preconditioning. In *Symposiun on Parallel Algorithms and Architectures (SPAA)*, 2008.
9. Koutis, I., Miller, G.L., Peng, R. Approaching optimality for solving SDD systems. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science, FOCS*, IEEE Computer Society, 2010.
10. Koutis, I., Miller, G.L., Peng, R. A near-m log n solver for SDD linear systems. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science, FOCS*, IEEE Computer Society, 2011.
11. Koutis, I., Miller, G.L., Tolliver, D. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. *Comput. Vision Image Understand. 115*(12) (2011), 1638–1646.
12. Krishnan, D., Szeliski, R. Multigrid and multilevel preconditioners for computational photography. *ACM Trans. Graph. 30*(6) (2011), 177.
13. Liben-Nowell, D., Kleinberg, J.M. The link-prediction problem for social networks. *JASIST 58*(7) (2007), 1019–1031.
14. Missiuro, P.V., Liu, K., Zou, L., Ross, B.C., Zhao, G., Liu, J.S., Ge, H. Information flow analysis of interactome networks. *PLoS Comput. Biol. 5*(4) (2009), e1000350.
15. Rudelson, M., Vershynin, R. Sampling from large matrices: An approach through geometric functional analysis. *J. ACM 54*(4), (2007), 21.
16. Sherman, J. Breaking the multicommodity flow barrier for $O(\sqrt{\log n})$-approximations to sparsest cut. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science* (FOCS '09, 2009), IEEE Computer Society, Washington, DC, USA, 363–372.
17. Spielman, D.A., Daitch, S.I. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, May 2008.
18. Spielman, D.A., Srivastava, N. Graph sparsification by effective resistances. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, 2008, 563–568.
19. Spielman, D.A., Teng, S.-H. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, June 2004, 81–90.
20. Spielman, D.A., Teng, S.-H. Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *CoRR*, abs/cs/0607105, 2006.
21. Tolliver, D.A., Koutis, I., Ishikawa, H., Schuman, J.S., Miller, G.L. Automatic multiple retinal layer segmentation in spectral domain oct scans via spectral rounding. In *ARVO Annual Meeting*, May 2008.
22. Trottenberg, U., Schuller, A., Oosterlee, C. *Multigrid*, 1st edn, Academic Press, London, 2000.
23. Vaidya, P.M. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. *A Talk Based on this Manuscript was Presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation*, October 1991.
24. Vassilevska Williams, V. Breaking the Coppersmith-Winograd barrier. In *Proceedings of the 44th Symposium on Theory of Computing*, STOC '12, 2012.

Ioannis Koutis (ioannis.koutis@upr.edu), Computer Science Department, University of Puerto Rico-Rio Piedras.

Gary L. Miller (glmiller@cs.cmu.edu), Computer Science Department, Carnegie Mellon University.

Richard Peng (yangp@cs.cmu.edu), Computer Science Department, Carnegie Mellon University.